Student Name…… Student ID…… Section……..

Question 1 [20 points]

You are given the following classes: `Item`, `Book`, and `BottleOfMilk`. The class `Book` and `BottleOfMilk` are subclasses of `Item`. Do NOT modify `Item.java`.

The abstract class `Item` has three common class variables as follow

```
double purchasePrice;    //Original purchase price in Baht
double age;   //Number of years this item has been purchased
double weight;                  //Weight of this item in kg
```

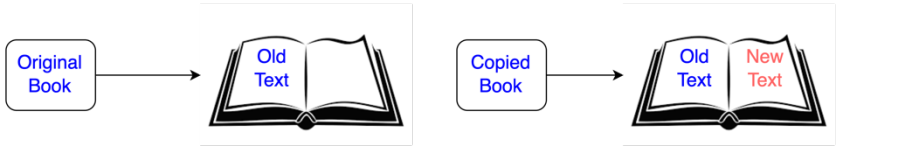
The getters, and `toString()` are already implemented. This class also contains two abstract methods: `getCurrentValue()` and `clone()`. These methods have to be implemented later for each subclass (i.e., `Book` and `BottleOfMilk`) based on its characteristics.

You need to implement the following methods in `Book` and `BottleOfMilk` class (by replacing **`//your code goes here`** comment in the `Book.java` and `BottleOfMilk.java`). A book allows you to take note on it, and its current value depreciates over time. A bottle of milk is consumable, and its current volume is what is left after you drink some of the milk. Specifically, let ***P*** be the original purchasing price, ***T*** be the age of the item, and ***C*** be the current value.

1. **`public double`** `getCurrentValue()` for `Book`.
   The current value of a book depreciates 10% each year. That is:
   $$C = P \cdot (0.9)^T$$

2. **`public double`** `getCurrentValue()` for `BottleOfMilk`.
   The current value of a bottle of milk is proportionate to its current volume. Let V_0 and V be the original volume and current volume, respectively. The current value is:
   $$C = P \cdot \frac{V}{V_0}$$

3. **`public`** `Book clone()` for `Book`.
   This method creates and returns a "deep" clone of this book. A deep clone is a process to create a new object by coping all the values from an original object. However, there is no link between original and new object after cloning. For example, after clone the original book, you will get a copied book. Both of them have a note "Old Text" inside. Later, you add a new note "New Text" in the copied Book, but no text will be added to the original book.



Old Text is copied from the original book.
BUT New Text only shows in copied book.

4. **`public`** `BottleOfMilk clone()` for `Book`.

This method creates and returns a "deep" clone of this bottle of milk.

Expected output from `ItemTester`:

```
Taking note: "Book 1"
Taking note: " -- Book 2"
Note on B1: Book 1
Note on B2: Book 1 -- Book 2
Drinking 50.0 ml of milk
Drinking 50.0 ml of milk
Milk left in m1: 250.0 ml
Milk left in m2: 200.0 ml
```

Files to submit: Book.java, BottleOfMilk.java

Question 2 [20 points]

You are hired to create a program to support FooBar café. This program can store and retrieve a menu of this coffee shop. The tester class called `MenuTester` in MenuTester.java is provided.

Task1 (5 pts): Create a class `MenuItem` according to the following UML diagram.

MenuItem
- name: String - price: double - calorie: int
+ MenuItem(_name: String, _price: double, _calorie: int) + MenuItem(_name: String, _price: double) + getName(): String + getPrice(): double + getCalorie(): int + increasePrice(int percent): int

This class has to store the following states of the menu item which cannot be accessed directly by other classes:

- String name: a menu item's name
- double price: a menu item's price
- int calorie: a menu item's calorie

Then, create two (overloading) constructor methods with the following signature

- `MenuItem(String _name, double _price, int _calorie)`
- `MenuItem(String _name, double _price)`

Both methods have to assign the given parameters to all attributes respectively. In the second constructor method, the program has to additionally assign the default value of 0 to the calorie attribute.

Task2 (8 pts): Create the following method

- `getName()`: to return this menu item's name
- `getPrice()`: to return this menu item's price
- `getCalorie()`: to return this menu item's calorie
- `increasePrice(int percent)`: to increase the current price by a given percent. For example, if the current price is 60 and the percent is 5, then the new price will be 60 + (5/100*60) = 63. Then this method returns a new price. [Note: If needed, you may create other methods. However, they will not be counted to your scores].

Task3 (7 pts): Complete a ***static*** methods in the provided `MenuTester` class

- `showOrderSummary (ArrayList<MenuItem> orders)`: to calculate total price and total calories of all orders, and then display the message in the following format

`Total Calories: XX, Total Price: YY`

Expected Output

```
------------------------------
       Coffee Menu
------------------------------
[1] Espresso: 60.0 (10 Kcal)
[2] Cappuccino: 80.0 (155 Kcal)
[3] Coffee Latte: 75.0 (150 Kcal)
[4] Coffee Mocha: 80.0 (340 Kcal)
[5] Water: 10.0 (0 Kcal)

Increase the price of Cappuccino and Coffee Latte by 8%
Cappuccino -> 86.4
Coffee Latte -> 81.0

Show the summary of the order
Total Calories: 155, Total Price: 96.4
```

Files to submit: MenuItem.java, and MenuTester.java

Student Name…… Student ID…… Section……..

Appendix: Class ArrayList<E> API

Modifier and Type	Method and Description
boolean	**add**(**E** e) Appends the specified element to the end of this list.
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(**Collection**<? extends **E**> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this ArrayList instance.
boolean	**contains**(**Object** o) Returns true if this list contains the specified element.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void	**forEach**(**Consumer**<? super **E**> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(**Object** o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(**Object** o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<**E**>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<**E**>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.

E	remove(int index) Removes the element at the specified position in this list.
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present.
boolean	removeAll(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection.
boolean	removeIf(Predicate<? super E> filter) Removes all of the elements of this collection that satisfy the given predicate.
protected void	removeRange(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
void	replaceAll(UnaryOperator<E> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	retainAll(Collection<?> c) Retains only the elements in this list that are contained in the specified collection.
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator.
Spliterator<E>	spliterator() Creates a *late-binding* and *fail-fast* Spliterator over the elements in this list.
List<E>	subList(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void	trimToSize() Trims the capacity of this ArrayList instance to be the list's current size.